TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ZEINAB ASHJAEI

LINUX KERNEL FUNCTIONS FOR AN EMBEDDED TARGET PLATFORM

Master of Science thesis

Examiners: Prof. Jari Nurmi
Dr.Tech. Tapani Ahonen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 1st September 2015

ABSTRACT

ZEINAB ASHJAEI: Linux Kernel Functions for an Embedded Target Platform
Tampere University of Technology
Master of Science thesis, 56 pages
April 2016
Master's Degree Programme in Information Technology
Major: Software Systems
Keywords: Operating system, Linux, Porting Linux kernel, Embedded systems, Linux Kernel, COFFEE RISC Core

In the earliest years of computer systems revolution in the 1930-40s, the computers were extremely expensive and huge, and they were dedicated to performing a single task or a collection of targeted tasks. Nowadays, the tendency of computer systems development is towards some small, fast, and very powerful tools, gadgets and equipment which have become part of our everyday life. These systems are called embedded systems. Although they were used only to control electromagnetically telephone switches at the beginning, their capabilities have improved gradually over the past decade. Obviously, this is a vital requirement for embedded systems to be able to connect to some networks in order to send and receive data. It could increase the level of complexity in embedded systems. Hence, they are required to have more memory and interfaces, as well as the services of an operating system to do memory management, network management, file systems and etc. Although there are many different kinds of embedded operating systems, the Linux OS is chosen in our case. Now the question is how the Linux operating system could be integrated into the embedded system hardware platform and make it compatible with the user applications.

If the target platform is one of the platforms already supported by the Linux, the porting procedures could be accomplished easily by using the codes and files provided by the Linux kernel. Otherwise, it is required to start coding from scratch. The target embedded system which is used in this thesis is called COFFEE Core. It is a RISC-based embedded processor that has been designed at Tampere University of Technology. COFFEE Core is considered as a general-purpose platform which is mainly designed for embedded systems. Since the COFFEE Core is not developed in the Linux kernel tree, it is required to integrate some pieces of code which should be written exclusively for COFFEE Core in Linux kernel tree. Accordingly, some modification in the hardware-independent sections is required.

Therefore, the main goal of this thesis is to illustrate what it means to porting Linux OS to a newly designed architecture. It provides a comprehensive programming

paradigm of the process of porting and explains how and in which order the porting could be fulfilled. Moreover, the architecture of Linux itself is presented and its different components will be reviewed.

PREFACE

The research work related to this Master of Science thesis is conducted in the department of Electronics and Communications Engineering, Tampere University of Technology, Finland.

I would like to express my great gratitude to my supervisors, Dr.Ahonen and Prof. Jari Nurmi for the given opportunity and their support, guidance and patience during my project.

My appreciation also extends to all my friends who were of great help and support throughout my whole education. Your friendship makes my life a wonderful experience.

My parents deserve a particular note of thanks: I am honored to have you as my parents. Thank you for giving me a chance to prove and improve myself through all my walks of life. I love you.

Finally, I would like to thank my lovely husband for his encouragements to take on this study and putting up with me for the past two years. If I ever lost interest, he kept me motivated. Thank you Farshad.

Tampere, 8.2.2016

Zeinab Ashjaei

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND SYMBOLS

ABI	Application Binary Interface
API	Application Programming Interface
ASP	Application Specific Processor
BIOS	Basic Input/Output System
BSP	Board Support Package
CCB	Core Control Block
CISC	Complex Instruction Set Computer
CPU	Central Process Unit
CR	Condition Register
DSP	Digital Signal Processor
FHS	File Hierarchy Standard
FIFO	First In First Out
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
GPL	General Public License
GPP	General Purpose Processor
GPU	Graphic Processor Unit
GRUB	GRand Unified Bootloader
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input/Output
IP	Internet Protocol
IPC	Inter Process Communication
LIFO	Last In First Out
MMU	Memory Management Unit
NTP	Network Time Protocol
OS	Operating System
PC	Personal Computer
PCB	Peripheral Control Block
PFN	Page Frame Number
PID	Process Identification
POSIX	Portable Operating System Interface for Unix
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing

ROM	Read Only Memory
RTOS	Real Time Operating System
SCI	System Call Interface
SJF	Shortest Job First
SPR	Special Purpose Register
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
TUT	Tampere University of Technology
URL	Uniform Resource Locator
VFS	Virtual File System
XFS	X File System

1. INTRODUCTION

In this chapter, the motivation of the thesis and its outline will be presented.

1.1 Motivation:

Although the modern world has been integrated with the computing power in every aspect of human life, there has been a new trend from traditional desktop computers towards hidden computing power which are known as embedded systems. An embedded system can be defined as a kind of computer which is designed for a specific purpose. They are widely integrated in devices from simple ones such as mp3 players to complicated systems deployed in process control, defense systems, telecommunication systems [29].

Despite the fact that there are embedded devices that are built efficiently without an operating system in the market, it is extremely rare to find an embedded system working without a real OS. In fact, the embedded systems perform a number of complex functions in complicated devices in medical equipment, industrial applications and consumer electronics. Therefore, we need to integrate an OS in embedded systems to organize and handle advanced functionality and make the embedded system more reliable and secure.

Operating System(OS) is a kind of software that acts as an interface between the hardware in lower layer of a computer system and the higher one which is related to user or applications. It is more convenient to use a computer by providing an interface for the user. The Operating system is responsible to manage the system resources, memory allocation, controlling input and output devices and so on [24]. In the last few years, by developing lighter operating systems, the usage of them in embedded systems has increased dramatically. Among various embedded operating systems, Linux has gained popularity due to its numerous advantages in the embedded system field. Linux is an open-source OS that makes it possible to be

modified and redistributed by vendors or developers freely. Obviously, the Linux kernel source tree should be compatible with the underlying hardware for the embedded system design to work properly. Since different platforms are different in their CPU's, memory interfaces, I/O buses, and their software binary standard requirements, the Linux OS code needs to be modified and recompiled to be able to run on every single platform. This procedure is called porting.

Earlier released versions of Linux operating system were extremely unportable and were written so that could run only on Intel 386 machines. The first attempt to make Linux portable started in 1993, when Linus Torvalds was offered to port Linux to Digital Alpha architecture. This project, however, took about one year and made Alpha architecture the second officially supported architecture in the Linux tree. This porting project has been performed by re-writing some pieces of the kernel to make it fundamentally portable. Then, the 1.2 kernel versions supported different architectures like MIPS, SPARC, and Intel x86. Currently, the number of supported architectures in the latest version of the Linux kernel, has increased to 20 with the addition of Motorola, IBM POWER, M32, x86-64, etc. Moreover, each of the mentioned architectures supports various chip and machine types as well [38].

In early 2000's, a group of hardware designers has developed a RISC-based multi-core processor, COFFEE Core, in Tampere University of Technology. Since it could not be categorized in one of the already supported Linux architectures, the Linux kernel should be ported exclusively for the COFFEE Core. The present thesis investigates the modifications needed to port a Linux kernel to make it run on this new platform.

1.2 Thesis Outline:

This thesis is divided into 6 chapters. Chapter 1 is an introduction part in which the motivation and structure of the thesis are provided. The second chapter focuses on Linux operating system and explains its components, architecture, and kernel sub-systems. Chapter 3 includes a short introduction of our target system, COFFEE RISC Core. Chapter 4 illustrates details about embedded Linux systems and their generic architecture. Moreover, Software elements of an embedded system are described. Chapter 5 is the implementation part of this thesis that clarifies the required steps to port the Linux kernel to a new architecture. The conclusion of the thesis is presented in chapter 6.

2. LINUX OPERATING SYSTEM, CONCEPTS AND ARCHITECTURE

Although Linux was targeted to desktop PCs at the beginning of its creation, nowadays Linux is the preferred operating system for almost all new embedded device projects like Internet appliances, telecom routers, switches, and automotive applications. Providing a robust, flexible kernel and run-time infrastructure, Linux can be ported to various microprocessors and platforms integrated into embedded systems.

Linux is a full featured open source UNIX system that has been developed for 80386 processor—the first true 32-bit processor in Intel's range of PC-compatible CPUs—by a Finnish student, Linus Torvalds in 1991 at the University of Helsinki. Since then, many open source communities contributed to the development of Linux voluntarily over the Internet to improve its features gradually. On the other hand, it has been extended by hardware vendors to support new processors, buses, devices, and protocols [36].

While choosing a proper OS running on a desktop computer, there are few options: Windows, Mac, or Linux. However, the matter is much more complicated for an embedded system because of the wide diversity of embedded applications. Some of the popular operating systems in Embedded field are QNX, VxWorks, Symbian and Embedded Linux [37].

Despite the real variety in the use of these operating systems, Linux gained market share among them and became the most popular OS in embedded system projects. A statistical analysis performed by UBM Tech Electronics shows that Linux is being used in more than 50 percent of embedded projects [10]. Also, this study identifies the main factors that influenced on Linux embedded developers' decision to use Linux in their projects. Although Linux is distributed under the GNU General Public License (GPL), according to the data in the figure 2.1, it is apparent that the key to the success of Linux is the availability of its source code and build tools

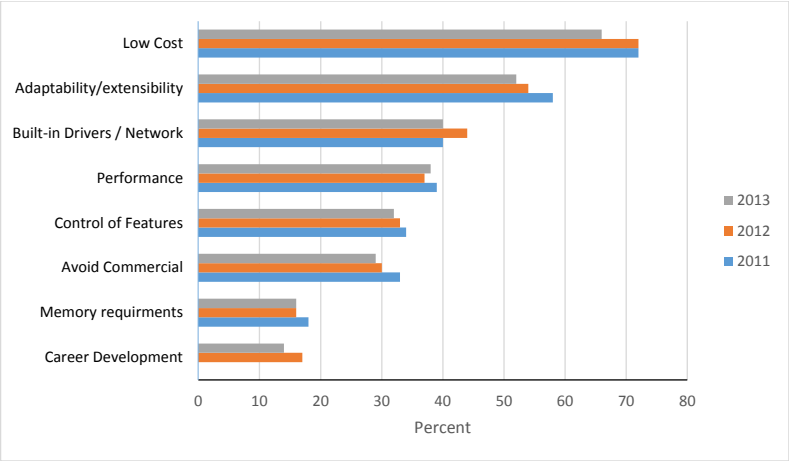

Figure 2.1 *Why are you interested in embedded Linux[10]*

which are accessible without any restriction.

Besides those mentioned items, Linux is famous for its exceptional networking capabilities which are an essential factor in embedded systems. Also, due to the modular nature of Linux kernel, it allows developers to remove utilities and system services that are not fundamentally in an embedded environment. However, the most interesting feature of Linux is its portability that provides the possibility to compile Linux and run it on a huge number of processors and platforms [29].

The next section, describes components of Linux systems and the structure of the Linux kernel in more details.

2.1 Components of Linux System

The Linux operating system is based on a layering structure that is comprised of three primary components:[36, 43]

1. **Kernel**: Is a piece of software written in C language that is considered as the central part of the Linux operating system. Technically speaking, it consists of different modules providing an abstraction layer so that it hides underlying hardware details from the system or running application programs. Therefore, the kernel itself is responsible for interacting with the hardware directly. Moreover, it contains many critical processes needed for the operating system to accomplish.

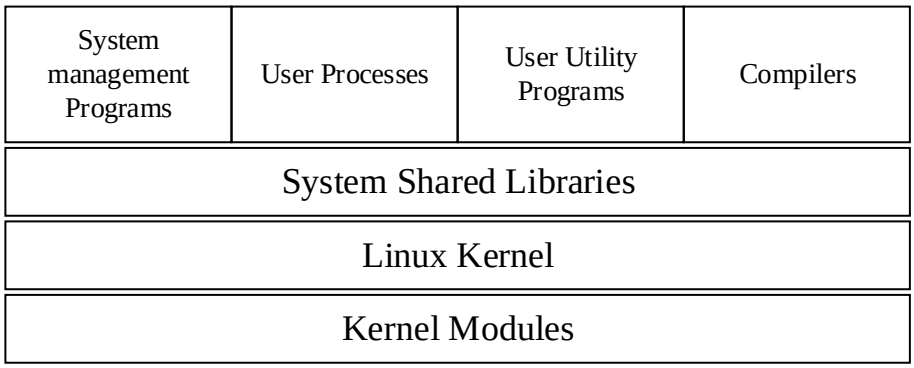

Figure 2.2 *Components of the Linux system [36]*

2. **System shared Library**: Is a standard set of functions through which application programs access the kernel's features. System libraries implement most of the functionality of the operating system and do not require kernel code privileges. C library, for example, is one of the most important system libraries available.

3. **System Utility** To be able to manipulate a Linux system, the kernel and system libraries are not sufficient. Rather then some utilities are required to access the commands or input given to the system that gets interpreted and executed. These vital utilities are called system utility that is defined as programs responsible for doing specialized, user level tasks. Some system utilities, for example, handle initialization and configuration of the systems or accept login requests from terminals and update log files, some others respond to incoming network connections or privilege related tasks and so on. Various Linux distributions use similar system utilities with the same features but different in their implementations.

Figure 2.2 illustrates the structure of a full Linux system consisting of various components discussed above. There are loadable kernel modules at the lowest layer of the hierarchy so that the Linux kernel can load them dynamically at run time. The Linux kernel is situated on the upper layer that includes all the necessary features of an operating system. The next layer belongs to the system libraries providing different types of functionality such as kernel system calls that will be discussed further in the next chapter. User mode or system mode programs that are called utilities are settled in the top layer of this hierarchy.

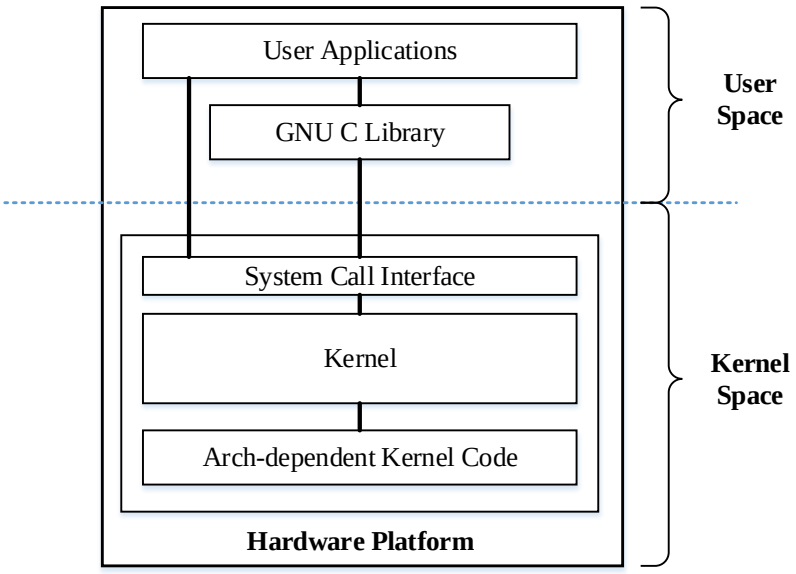

Figure 2.3 *General Architecture of a Linux System [46]*

2.2 Architecture of the Linux Operating System

A Linux-based operating system structure is in much the same way as other operating systems. It consists of different layers which have their own functionality and special mechanism to make them communicate with each other. Fundamentally, when Linux is running, it could be divided into two parts. Indeed, Linux can run either in User Mode or Kernel Mode. User mode and kernel mode occur in the user space and kernel space respectively, and represent two distinct address spaces [3].

Figure 2.3 represents the overall architecture of a generic Linux system. As can be seen, on the top of hardware platform there is the kernel space comprising architecture dependent kernel code. It enables Linux to operate on a vast array of hardware platforms. Furthermore, the kernel itself and system call interfaces are located in the kernel space too. The user space, belonging to user applications, resides above the kernel space. There is also the GNU C Library that acts as an interface between the user space applications and the kernel. Also, the lowest level refers to the hardware platform. This layer includes system physical equipment from visible ones like the monitor, keyboard, mouse, graphic or network card to non-obvious parts such as the CPU or RAM in the system.

In this section, user and kernel mode and also the work-flow between them will be discussed briefly.

2.2.1 Kernel Mode:

All the kernel code is executed in a privileged mode with full access to physical resources of the system. It is called as Kernel Mode. In this mode, the kernel provides protected access to processes and invoke their required system services to accomplish their procedures.

Each kind of CPU defines a special mechanism that allows processes to switch from user mode to kernel mode and vice versa. Before discussing their interaction procedures, it is necessary to describe **system calls**. Linux has implemented a set of interfaces for user programs so that they could access hardware layer resources and also other operating system services. These interfaces are called system calls. System calls range from the familiar functions such as read() and write(), to the exotic, like sigaltstack() or getresuid().[3]

The system calls fulfill two main purposes:[22]

- System calls allow user space applications to interact with the hardware without concerning about the different types of low-level devices by providing an abstract hardware interface. For instance, when reading or writing to a file, the user space application does not care about the type of underlying disk, media or file system, rather invokes the related system calls instead, and they perform the reading or writing tasks.
- System calls enhance system security and stability. As long as the kernel resides between the hardware resources and user space applications, it can inspect processes based on their permission, the kind of user or other criteria. Hence, the kernel could prevent any harm to the system.

2.2.2 User Mode:

The user mode is defined as the protected space in the memory where the user applications or system programs are executed. In this mode, direct accesses to system hardware, kernel programs or kernel data structures are prohibited. Instead, they use system libraries to access kernel functions which are reside in lower levels of operating system.

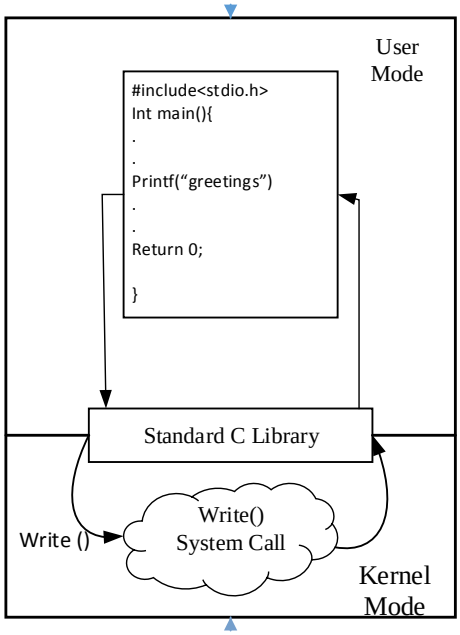

Figure 2.4 *The relationship between application, C library, and the kernel when calling printf()*

Generally speaking, a user space application uses an Application Programming Interface (API) instead of calling system calls directly. In a simple term, Linux provides a standard API around the system calls to make them convenient to use. The caller does not have to know how the system call has been implemented internally. What are important for the caller to work correctly are the format of the API and the return value of the system call. In Linux world, APIs follows the POSIX standards which are composed of a set of IEEE standards. APIs in a Linux system are implemented based on the C library, including standard C library functions and system calls.[2, 22]. Figure 2.4 depicts the role of a C Library and the relationship between the user space application, C library, and system call when calling printf () function in user space. The printf () invokes its counterpart in C library that is printed. Then the C library converts it to write () function. finally, the write function of the system call is called to fulfill the print () command received from the user space.

2.2.3 Interaction between the user and kernel space

In Linux operating system mechanism, the procedure of executing system calls comprises of 4 stages:[48]

1. While a user application invokes a system call; an interruption is produced to inform the kernel about the related system call. Each system call is associated with a number. All the assigned numbers are maintained in an indexed table to retrieve more easily. The user space program puts the number of the respective system call in a special register called eax before context switching to kernel mode.

2. The system switches to kernel mode and control is taken over by the kernel. The specified system call is obtained from the eax register by using sys_call_table. Then, the related entrance address is loaded.

3. The kernel jumps to the system call entry, and it executes the hardware-dependent CPU instruction on behalf of the user space program.

4. Finally, the system call ends and the system returns to user space to resume the running program.

In the next section, the kernel will be explored from the functional point of view, and some of the most critical subsystems of the Linux will be reviewed briefly.

2.3 Components of the kernel

As explained earlier, the kernel is considered as the heart of the operating system whose responsibility is shortened as a resource manager. Whether the resource is a CPU, memory, or Input/Output devices, it manages and mediates access to the resources of the system between multiple user applications that are competing to consume resources.

The Linux kernel provides major services for all parts of the operating system that are required by either other parts of the OS or the user applications. These services are process management, memory management, device management, handling file systems and networking. The kernel also provides methods for synchronization and communication between processes called inter-process communication or IPC [45].

Figure 2.5 shows the internal architecture of the Linux kernel. As can be seen, the kernel could be divided into five different subsystems. Each of them performs a defined functionality and offers it to other subsystems. This configuration is also evident in the kernel source code, where each of these subsystems is written in their sub-trees.

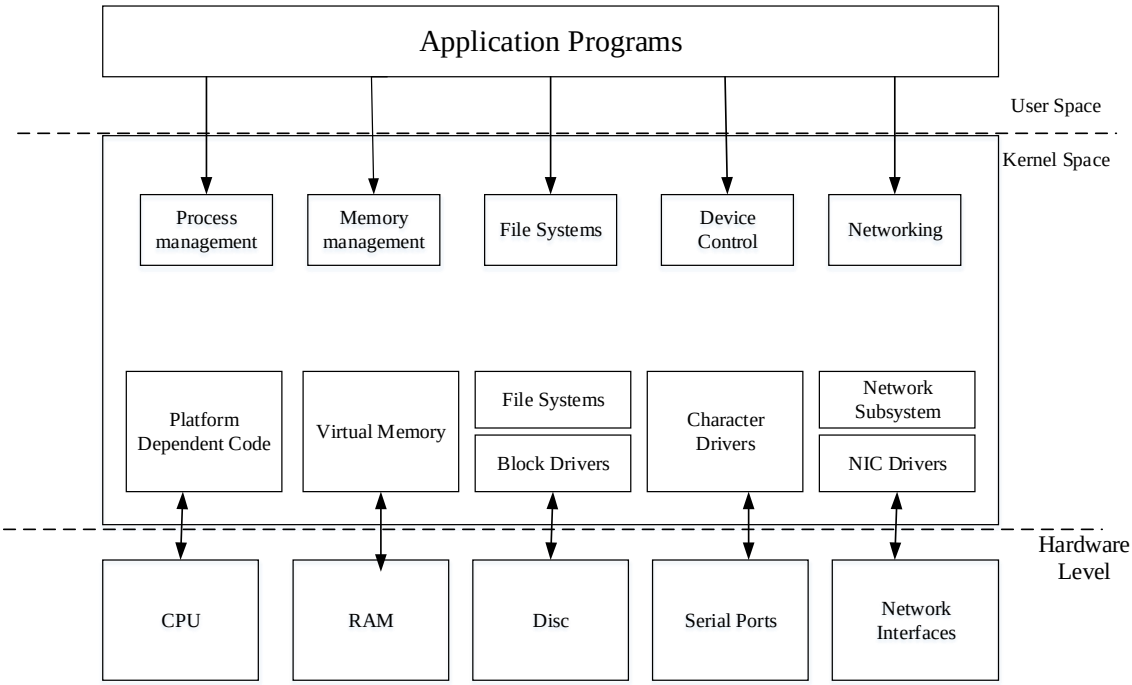

Figure 2.5 *Architectural Perspective of the Linux Kernel*

2.3.1 File Systems

The Linux operating system has a unique integrated approach to its design that is based on files. It is worth to know that in Linux, everything is configured as a file. The concept covers not only text files, images or compiled programs, but also directories, partitions or hardware device drivers [7].

File:

As illustrated above, everything is treated as a file. Conceptually, files are nothing more than a data container structured as an ordered string of bytes. To be more accurate ,"A file is a named collection of related data that appears to the user as a single, contiguous block of information and that is retained in storage"[16]. To the user, files could be seen in a tree-structured name-space, as shown in Figure 2.6. Each file is distinguished by a name that is unique within the directory in which the file is located.

Unlike in the structure of Windows that has separate starting points for each file or device, there is just one source for everything in Linux. As can be seen in the figure above, this is the top-level directory of the tree which is called the root or Slash (/).

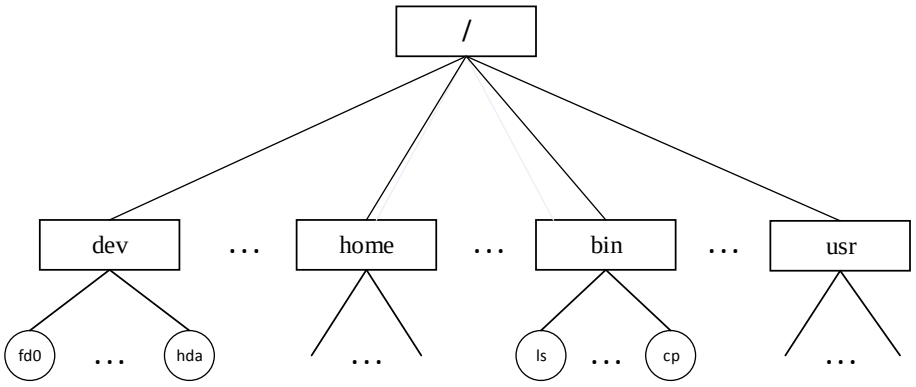

Figure 2.6 *A directory tree in Linux*

Everything else in the file tree is considered as the sub-directory of the root that each of them, in turn, comprises further sub-directories. This file system structure is called the Filesystem Hierarchy Standard (FHS). The major directories of FHS are:

- /bin, command binaries for all users
- /boot, boot loader files such as the kernel
- /home, users home directories
- /mnt, for mounting disk storage
- /root, home directory for the root user
- /sbin, executables used only by the root user
- /usr, where most application programs get installed

File Descriptor and Inode:

From the kernel point of view, the FHS is flat and files are identified based on a unique number rather than their name or location in directories. In fact, the kernel uses inodes to represent each file. An inode is an entry in a list of inodes that contains the information required to access the file including:[12, 7]

- The inode number which is associated with a specific file

- The owner of the file which is a user or a group of users
- The type of the file (directory, device, pipe, ...)
- The file creation, access and modification times
- The size of the file
- The pointer to data blocks that store the file's content

FileSystem

A file system is a collection of methods and data structures that are used to organize files on a disk drive (hard disk, floppy disk, CDROM, etc.) In other words, the operating system uses file systems to keep track of files on a disk or partition. The responsibility of the file system is to store, retrieve or update data on a disk. The file system is a very critical part of the kernel since it must hold data safely and securely to avoid lost data and files.

Nowadays, Linux supports various file systems usable in different architectures such as ext, ext2, ext3, ext4, hpfs, iso9660, JFS, minix, msdos, ncpfs nfs, ntfs, proc, Reiserfs, smb, sysv, umsdos, vfat, and XFS [20].

Virtual File Systems

The Virtual File System (VFS) is a software abstraction layer between the user applications and the supported file systems in the Linux operating system. VFS is one of the most interesting features of Linux that makes it flexible, so that it allows Linux to support many, often very different file systems simultaneously. The user applications can access different file systems on various devices without any worry concerning their structure or detail. It could be achieved by using a set of Linux file systems [14]. Figure 2.7 depicts a simple diagram of VFS and the associated file systems in a kernel architecture.

A user application request such as open, close, read or write manipulates the data stored in different devices that may use various file systems. The system call interface (SCI) obtains those commands and dispatches them to the VFS layer. Each of the file systems has defined the implementation of the upper-layer functions exclusively. Below the file systems, there is the buffer cache layer that includes a common set of functions that are applicable to all kinds of file systems. They are independent of any

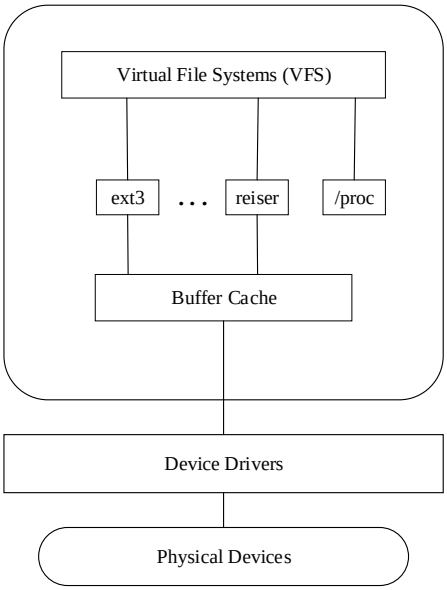

Figure 2.7 *The VFS relation with the file systems*

single file system and help the OS to optimize access to the physical devices through storing data for a limited time. Finally, device drivers operate as an interface to the physical devices layer.

2.3.2 Process Management

Process management is the most crucial part of the operating system whose responsibility could be classified into three sections. First, it creates and destroys processes and handles their interaction with I/O peripherals. Besides, the communication between various processes is performed by the process manager. Finally, it divides the processor time between various processes on the system. This functionality is called process scheduling [8].

This section introduces the concept of processes and examines the mentioned three responsibilities of the process management.

Process:

A process can be defined as an instance of a running program in memory. A program, however, is an executable file including a collection of machine code instructions and their allocated data structures. While running a program, those instructions

are copied into the memory, and the required space is allocated for the program's variables. Since processes run in their individual address spaces, processes could not interfere with each other and cause other's processes to crash [23].

Process's system calls

The kernel provides system call interfaces to create, execute or stop a process and manipulate them. The act of creating a new process is called forking which is performed by *fork()* system call. It duplicates the current running process. After creating a new process, it is desirable to execute a new program. It could be accomplished by *exec()* family of system calls that is responsible for creating a new address space and loads a new process into it.

A process is terminated through invoking *exit()* system call. It closes the process and frees all its allocated resources [23].

Task_Struct

The Linux process management unit is responsible not only to observe process's activity during their lifetime, but also controls system resources used by processes effectively. Indeed, a process uses the CPU and physical memory to execute its instructions and store data respectively. Also, it interacts with files within a file system and physical devices on the system. Therefore, there is a significant amount of data regarding each process that should be recorded precisely to enhance the performance of the system when retrieving information of processes. The name and owner of the process, the address of the memory allocated to the process, the list of files or other I/O streams that it has opened during execution, and the state of the process are some of the categories that are organized in *task_struct* data structure [13, 23].

Each process is designated by a task_struct data structure whose fields describe all information about a particular process. Its main fields are as below:

- **State:**

  This item determined the execution state of the process and illustrates what is happening to a process at the moment. As can be seen from figure 2.8, each process state is one of 5 different states as follows:

  - **Running:** As its name implies, a running process corresponds to two

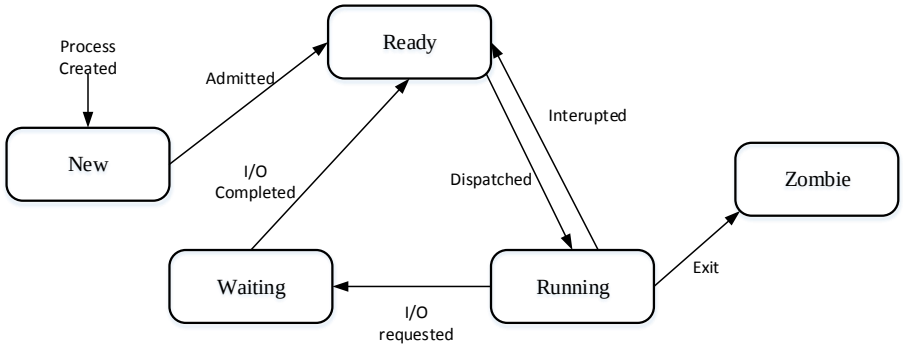

Figure 2.8 *State of processes flow chart*

states. It is either executing as the current process of the system or is in the run queue waiting to run. A running process is the only possible process that can be executed in user space.

- **Waiting:** A process is in waiting state if it waits for a condition to be reached, or a resource to be available. The waiting state could be divided into two sub-states; Interruptible and Uninterruptible. In the latter state, the waiting condition is related to the hardware condition directly, whereas interruptable processes could be interrupted by other process's signals.
- **Stopped:** The process execution has been stopped and it is not running anymore. A process goes to this state usually by receiving a signal. In addition, a process that is being debugged can be put into the Stopped state.
- **Zombie:** A zombie process is a dead process whose task structure, for some reason, is still remaining in the task vector.

- **Process ID:**

  While creating a process, a unique numerical value is assigned to it referred to as a process identification (PID). It is used by the operating system to refer to that process. The PID of 1 is dedicated to the init process that is always the very first running process when booting the operating system.

- **Links:** In the Linux system the processes are not independent of other processes, rather there is a distinct hierarchy between them so that each process could spawn (give birth to) other processes by creating an identical copy of

itself. The new process, in turn, could produce another process, thereby resulting in multiple levels of processes. Therefore, it is sensible that each process has one parent and one or more children. Obviously, if a process dies, all its children dies respectively. The *Link* field keeps a pointer to each process's parent, siblings and its children. Consequently, given the current process, it is possible to obtain the process descriptor of its parent or children by using pointers.

- **File system:**

  As described earlier, processes may interact with the files and try to open or close them. The *file system* field holds pointers to each open file, also pointers to the root and current directories for the process. In this way, it is not allowed to delete the directory that a process is referring as its current directory or sub-directories.

- **Address space:** Defines the virtual address space assigned to this process.

Inter Processes Communication:

Although processes are non-sharing entities, we sometimes need to make them communicate with each other, especially in a multi-programmed and network environments. Inter-Process Communication (IPC) is a provided operating system mechanism that allows exchanging data between two or more independent processes. The Linux kernel supports uni-directional, bi-directional or multi-directional process communication [5]. IPC mechanism can be implemented in some different ways that are chosen based on system requirements or its flexibility. Some of them are listed as below:

- Pipe: Allows the data flow only in one direction between processes
- Message queues: message passing using queues
- Shared Memory: Allows data flow between processes by defining a section of memory as shared memory

The IPC is typically conducted by *message passing* mechanism in which a process (sender) would request to send a message to another one (receiver). In this way,

that message is stored in a special message queue, and then the receiver could access it by invoking a particular system call.

Scheduler:

Since the current version of Linux kernel supports multi-tasking concept, multiple processes could exist simultaneously in the system. Therefore, it is vital to share the processor time between them as fairly as possible. The process scheduler, however, is integrated into the Linux kernel to divide resources among all running processes so that they are given the illusion that they are the only processes in the system. Moreover, the scheduler determines the next process to run by considering kernel scheduling policies and assigns a time slice to be executed at a particular time [44].

Basically, scheduling procedure in Linux is based on the time sharing technique in which the processor time is divided into time slices called quanta. Each process is given a quanta for accomplishing its task so that the length of quanta is calculated according to the importance level of processes. However, if the running process could not terminate throughout its time slice, it waits and the processor switches to another process which is nominated as the next process in the processor run queue. In fact, the CPU applies a suitable scheduling algorithm for selecting subsequent processes to run. Some of the famous scheduling algorithms are: [30, 22]

- First-In-First-Out (FIFO)
- Shortest-Job-First (SJF)
- Last-In-First-Out (LIFO)
- Multilevel queue scheduling
- Real Time Scheduler
- Round Robin Scheduler

2.3.3 Memory Management

Since the memory is the most valuable resources within a computer system, it is necessary to review Linux memory management layout briefly. Although the size of memory devices has increased drastically in recent years, so has the size of the

application and data to be processed. Memory allocation should be managed to prevent unexpected faults regarding memory in the system. Memory management subsystem observes each memory location in the system, whether they are allocated to the processes or they are free. Moreover, it decides the amount of required memory that should be provided for each process to use in a determined time [11].

In this section, a short overview of the most important aspects of memory management is given; physical and virtual memory, swapping, fragmentation, and paging.

Physical Memory vs Virtual Memory

Traditionally, physical memory is a block of memory that exists in the system physically. It records data with low latency in the RAM of the system. DRAM, SD memory card, video cards, network cards, are examples of a physical storage hardware that are frequently used. A *Physical memory address* is a binary number across a memory bus that corresponds to a memory cell within a storage device. In contrast, there is the virtual memory that has emerged in modern operating systems to overcome physical memory limitations. Implemented in both the hardware and software, virtual memory helps the operating system to run processes that need more memory space than actually available. The virtual address space is provided for each process so that each process thinks it is the only running one in the system and the whole memory space is belongs to it. The virtual and physical address spaces are divided into fixed length slots called pages that are defined as the smallest addressable unit of memory management unit. Pages in virtual address space are called virtual pages, whereas, in physical memory, they are known as page frames that are referred to by page frame numbers (PFN). Indeed, a frame is a place where a page is placed physically [9].

Using a special data structure called page tables, the memory management unit (MMU) maps the virtual address ranges to their associated physical addresses. Note that only the kernel accesses physical memory addresses directly. On the other hand, the user-space applications use virtual addresses exclusively [4].

Figure 2.9 shows the translation procedure between virtual and physical addresses via page tables. Linux OS prepares a set of page tables for each process that contains a process's list of memory mappings and tracks the associated resources.

Swapping

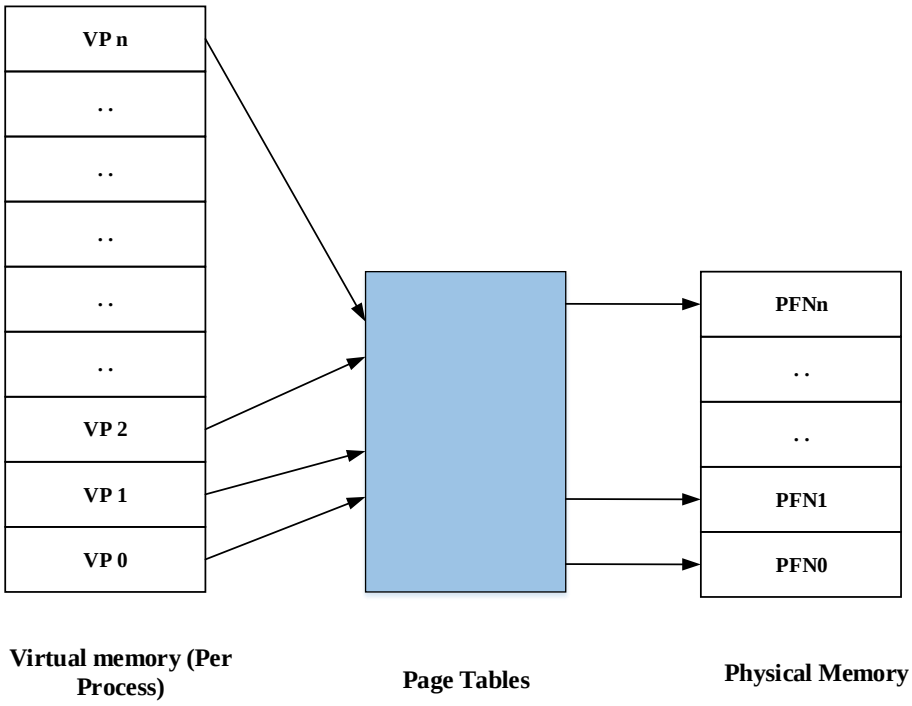

Figure 2.9 *Conversion of virtual and physical memory in Linux memory architecture*

In some situation, the physical memory overflows due to a large number of active processes or unexpected growing of a process in the system memory. In fact, a needed page frame should be settled in physical memory, but there is no free space. Linux provides a mechanism to release space in memory called swapping [4].

Swapping transfers those pages that are not immediately needed from physical memory to secondary storage temporarily. Then they might be brought back into the memory to continue execution whenever a process requires them. Therefore, when all of the physical memory is being used, swapping mechanism increases the total memory effectively.

Fragmentation and Paging

Fragmentation is a kind of a problem that occurs in a memory allocation procedure. Figure 2.10 illustrates the two variants of fragmentation; external and internal fragmentation [31].

- **Internal:** Occurs when a memory block that is larger than necessary space is granted to a process. Therefore, a portion of the memory block remains unused; also it is not usable by other processes.

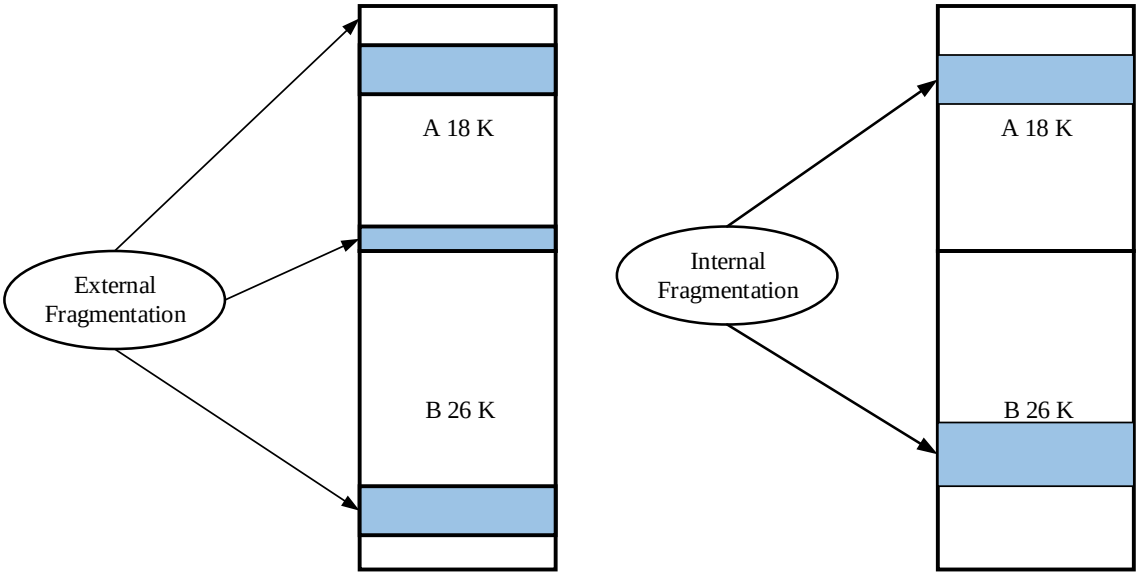

Figure 2.10 *External Fragmentation and Internal Segmentation*

- **External:** Although the enough memory may be free overall, memory allocation is not satisfied. Because the free space is not contiguous to reside a process on it.

Paging is a technique through which external fragmentation is avoided. As explained earlier, physical memory is divided into page frames whose size depends on the system architecture. While executing a process, its corresponding pages are loaded into any available page frames. In this way, virtual address space of a process can be non-consecutive; in turn, the allocated physical memory space is wherever the free memory frame is available. The Linux kernel keeps track of all allocated spaces by the specific process [21].

2.3.4 Device Drivers

The majority of Linux source code exists in device driver source tree. Also, all system processes map to a physical device eventually. Therefore, the correctness of device drivers is essential to the usability of the whole operating system [26].

A computer system consists of many kinds of devices from CPU as the brain of the system to general types such as storage devices, network devices, and human-interface ones like mouse, keyboard, and screen. The Linux source tree provides a

sub-directory for all devices that are supported by the kernel. Generally speaking, the device driver is defined as an abstract layer between the software concept and the hardware device that provides a standard interface to higher level applications, so that it hides the details of how the device operates. In this way, standard calls are independent of the specific hardware device and device driver sub-system maps them to their respective device-specific operations. Every device in a system supports 6 basic operations within the VFS; Open, Close, Read, Write, Seek, and Tell. It is interesting to note that device drivers can be developed separately from the kernel and plugged in run-time whenever needed [8, 42].

2.3.5 Networking

Linux is considered as a child of the internet since Linux developers have been using the web to spread their ideas and exchanging information through Linux communities. So, networking is an intensive Linux sub-system that offers all the necessary networking tools and features to integrate with all kinds of network infrastructure.

The network sub-system is an abstract layer whose role is to provide a mechanism for network connectivity between Linux systems and other machines. It supports many hardware devices, also network protocols such as TCP/IP. Network layer, like other abstract layers in Linux, hides the implementation details of existing devices and protocols so that processes or other kernel subsystems interact with the network layer without knowing about the underlying physical device or protocol that is being used [45, 49].

The network design in Linux obeys a layered architecture model consisting of three layers as shown in figure 2.11: [34]

1. The top most layer is SOCKET interface layer that is a standard API invoked through system call interfaces. The socket layer determines the appropriate network protocol and sends the system call to that corresponding protocol implementation.

2. The middle layer is called the protocol layer that consists of transport and network layer protocols. The Internet Protocol(IP), however, is the core network layer protocol. Similarly, the Transmission Control Protocol(TCP) is the main transport protocol family inside the protocol layer. Besides, it is possible

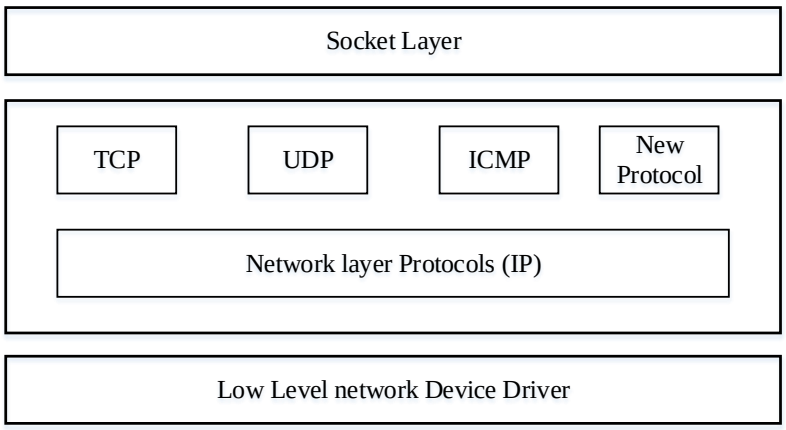

Figure 2.11 *Top level view of Linux Network Sub-system*

to plug in a new protocol family as a separate module and make the system compatible with it.

3. The lowest level belongs to the network device driver that provides access to the physical devices.

3. PLATFORM ARCHITECTURE

The target platform that is used in this thesis is an open source embedded processor called COFFEE RISC Core. COFFEE is developed at the Department of Computer Systems in Tampere University of Technology in Finland. The COFFEE Core user manual is accessible on the official web page[1] of this processor.

Since a processor in a computer system is responsible for performing all the computing processes (calculation, comparison, and logical decision), it is referred to as the brain of the computer system. Accordingly, the processor forms the most important part of an embedded system. Therefore, choosing the appropriate embedded processor is critical to perform the expected operations to guarantee the success of the whole system. But which kind of processor is suitable for our embedded system? The most important types of an embedded system processor can be divided into following items: [19]

1. General-Purpose Processor (GPP)
    - GPPs are designed to execute multiple applications and perform multiple tasks so that the end-user can program them to perform a broad range of different applications.
    - Usable in personal computers, workstations, mainframes or servers
    - They have the low prices, short design time but low performance and high energy consumption
2. Application-Specific Processor (ASP)
    - ASPs are designed to perform a particular set of on-demand applications and cannot be efficiently operational for other applications than the one they are designed for.

[1] http://www.coffee.tut.fi/documents.html

- Usable in home appliances, consumer electronic devices, networking, and communications, etc.
- They have good performance, small and less energy consumption, but more delicate design, more expensive and less flexible

Besides power consumption, computing performance and flexibility that are the main categories that should be considered when developing embedded systems; the final cost has to be evaluated too. Hence, adding additional resources with no value is not desired. Consequently, when designing an embedded system, a critical issue is to get a good match between the application demands and hardware resources [41, 27].

3.1 Coffee RISC Core Overview

As described in the previous section, according to our requirements, we can use many kinds of processors in embedded systems such as general-purpose or Application-specific. Although application-specific ones fulfill the lower power consumption and higher performance rate, they have the drawback of being less flexible for future new applications. Nowadays, the new trend is towards integrating several processors, also different kinds, in the embedded systems. There could be a general-purpose main processor and, for example, a GPU for graphics processing, application-specific core(s) or a DSP for signal processing, etc. The central processor handles controlling the whole system and off-loads a process to a suitable type of processor when needed.

The COFFEE RISC core is a general-purpose embedded processor core that has been designed mainly for telecommunication and multimedia applications so that coprocessors could be accelerated in computationally intensive tasks if needed. In this chapter, some of the most important specifications of COFFEE RISC Core regarding software and hardware will be introduced in more details.

3.1.1 Instruction set

While introducing IBM 360 in 1964, instruction set was defined as "the structure of a computer that a machine language programmer must understand to write a correct program for that machine". Instruction set acts as the interface between hardware

and software so that it provides a structure by which the software could communicate with the hardware to tell what should be done. Instruction set operations could be classified as arithmetic and logical, data transfer, control, system, floating point, decimal, string, multimedia or vector.

There would be three kinds of instruction sets in computer systems:

- Reduced instruction set computer or RISC
- Complex Instruction set computer or CISC
- Digital Signal Processing instruction set or DSP

The first two ones are suitable for general- purpose and the third one for application-specific processing. In COFFEE RISC core, the philosophy of RISC instruction sets has been adopted in order to drive a right processing engine for embedded systems. Although the minimum set of instructions has been implemented in COFFEE Core, the assembly language interface can be extended with pseudo instructions [1].

COFFEE has 66 instructions in its implementation, including, fourteen arithmetic instructions, ten bit-field manipulation instructions, six boolean instructions, eight conditional jumps and so on. The complete list of COFFEE instructions has been illustrated in table 3.1. In addition, instructions that process the data operate on two register operands, or, one register operand and one immediate operand and write the produced data to any general-purpose register.

3.1.2 Processor Operating Modes

Forasmuch as one of the particular specifications of COFFEE Core is to support real-time operating systems (RTOS), two modes of accessibility have been designed with COFFEE RISC Core; Supervisor mode and User mode. In Supervisor mode, the whole memory space and both register banks are accessible, whereas in user mode, access to protected memory areas is denied, and only the first register bank is available. Furthermore, designers of COFFEE have provided the possibility to switch from supervisor mode to user mode, also from user mode to supervisor with the help of system calls and trap instructions that are responsible for transferring the control to the supervisor mode that is the operating system in most cases.

Table 3.1 *COFFEE RISC Core instruction set.[27]*

Integer Arthmatic			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
add	Add 32-bit integer in registers	muli	Multiply 32-bit register with 16-bit immediate
addi	Add 32-bit integer with 16-bit immediate	muls	Multiply signed 32-bit integer (in registers)
addiu	Unsigned version of Addi	mulus	Multiply unsigned 32-bit integers(in registers)
addu	Unsigned version of Add	mulus	Mixed unsigned-signed multiplication
sub	Subtract 32-bit integers	muls_16	Multiply signed 16-bit integer (in registers)
subu	Unsigned version of Sub	mulu_16	Multiply unsigned 16-bit integers(in registers)
mulhi	Get upper 32 bits of 64-bit multiply result	mulus_16	Mixed unsigned-signed 16-bit multiplication
Byte and bitfield manipulation			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
exb	Extract(immediate-specified)byte from word	muli	Multiply 32-bit register with 16-bit immediate
exbf	Extract bitfield from word(register mask)	muls	Multiply signed 32-bit integer (in registers)
exbfi	Extract bitfield from word(immediate mask)	mulus	Multiply unsigned 32-bit integers(in registers)
exh	Extract halfword from word	mulus	Mixed unsigned-signed multiplication
lli	Load lower halfword from immediate	muls_16	Multiply signed 16-bit integer (in registers)
Boolean bitwise operations			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
and	AND two 32-bit register values	or	OR two 32-bit register values
andi	And 32-bit register with 16-bit immediate	ori	OR 32-bit register with 16-bit immediate
not	Bitwise NOT for a 32-bit register value	xor	Bitwise XOR two 32-bit register values
Conditional jumps (branches)			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
bc	Branch if carry	bgt	Branch if greater than
begt	Branch if equal or greater than	blt	Branch if less than
belt	Branch if equal or less than	bnc	Branch if not carry
beq	Branch if equal	bne	Branch if not equal
Other jumps			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
jal	Jump (based on immediate offset) and link	jmp	Jump (based on immediate offset)
jalr	Jump (on register) and link	jmpr	Jump (on register)
Integer comparison			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
cmp	Compare registers (set flags)	cmpi	Compare register to immediate (set flags)
Shifts			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
sll	Logical shift left	srai	Arithmetic shift right based on immediate
slli	Logical shift left based on immediate	srl	Logical shift right
sra	Arithmetic shift right	srli	Logical shift right base on immediate
Memory load, Store, Move			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
ld	Load word	mov	register-to-register move
st	Store word		
Coprocessor instructions			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
cop	Coprocessor instruction	movtc	Move data to coprocessor
movfc	Move data from coprocessor		
Mode changing instructions			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
chrs	Change register set	reti	Return from interupt
di	Disable interupts	retu	Return to user mode
ei	Enable interupts	scall	System call
swm	Switch decoding mode		
Miscellaneous			
Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
rcon	Restore condition from GP-register	trap	software exception (programmed interupt)
scon	Save condition in GP-register	nop	No operation, idle

16 and 32-bit decoding modes: The COFFEE RISC Core operates on 16 and 32-bit decoding mode that refers to the length of the instruction word. Although switching between these two modes could be done by using swm instruction, there is some limitation in 16-bit mode that should be considered. For instance, the conditional execution, instructions lui, lli, exbfi, and cop are not available. Moreover, immediate constants are shorter, and only eight registers in each set are accessible.

3.1.3 Registers

By considering the fact that the COFFEE Core is a load-store machine, the precise explanation of the register's structure becomes more important. As COFFEE Core follows the RISC instruction set architecture, a large bank of registers is designed in order to avoid excessive memory traffic. The COFFEE Core programmer's view of the register set can be in Figure 3.1. This register bank is divided into two separate general purpose register sets so that the first one is dedicated to user mode programs, whereas, the second one is specific to the privileged or supervisor mode applications such as an operating system. This system of classification was developed for the purpose of implementing user mode and supervisor mode in real-time operating systems (RTOS) and enable context switching between them as well.

Each of the register set contains 32 registers including general purpose registers (GPR) and specific purpose registers (SPRs). Furthermore, eight condition registers (CRs) have been provided in COFFEE that are usable by conditional branches. Beside these registers, there are two more blocks called Core Configurable Block (CCB) and Peripheral Configurable Block (PCB) which are provided to enable the desired software configure ability and controlling the processor operation.

3.1.4 Interface of the core

The type of COFFEE Core memory interface is Harvard in which program instructions are stored in different memory from data. In this case, each type of memory is accessed via a separate bus, allowing instructions and data to be fetched in parallel.

An example of the COFFEE Core interface can be seen in figure 3.2. There would also be other possibilities to connect to the core as well. From the figure, we can see that up to four coprocessors could be situated in COFFEE Core so that their

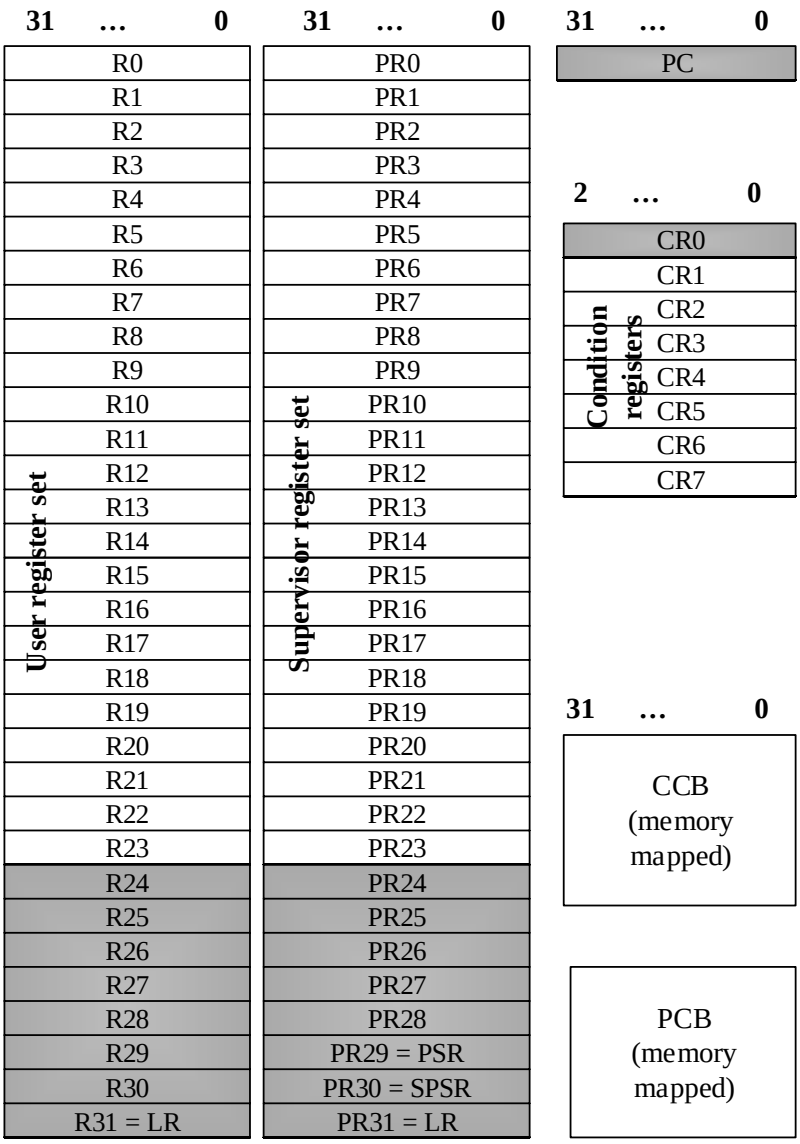

Figure 3.1 *The programmer's view of COFFEE Core register sets*

maximum register bank size is 32. Acting as a memory interface, coprocessors addressing is limited to 7 bits including two bits for coprocessor identification and 5 five bits for coprocessor register indexing. On the other hand, the PCB register block is also connected to COFFEE Core's data bus interface. While data memory accesses assert the general write (wr) and read (rd) signals, peripherals would assert PCB signals that are `pcb_wr` (Peripheral Control Block write) and `pcb_rd` (Peripheral Control Block read) instead of data memory.

Moreover, coprocessors are designed so that could interrupt core by asserting an ex-

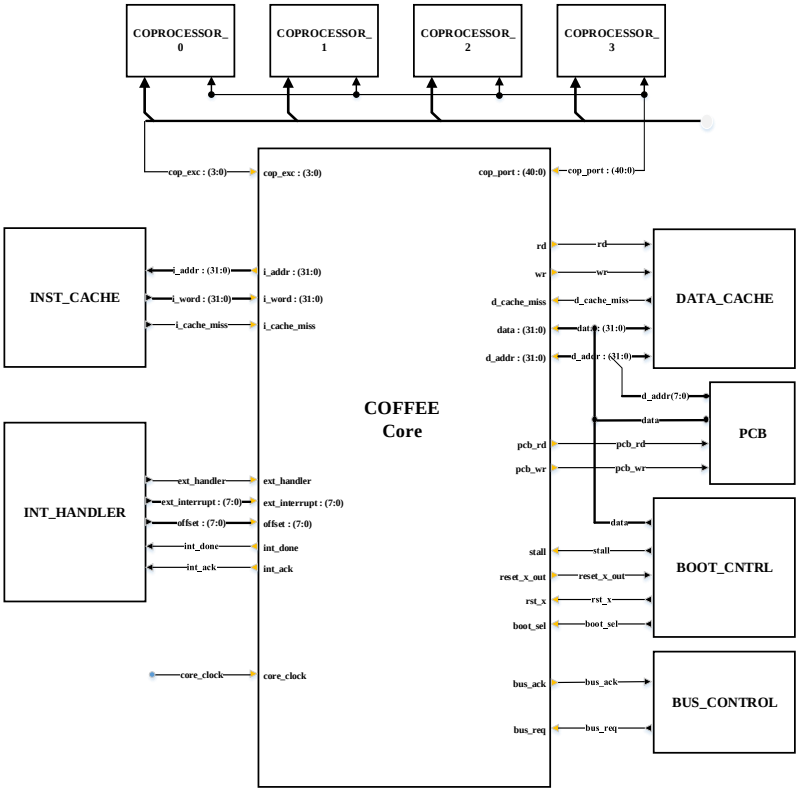

Figure 3.2 *Interfacing the COFFEE RISC Core*

ception signal. However, eight external interrupt sources is supported by COFFEE Core directly, it might be possible to use disconnected coprocessors exception signaling as interrupt request lines. As a result, it is potential to connect twelve interrupt sources to the core. Note that priority for coprocessor exceptions or interrupts is always set by software.

4. EMBEDDED LINUX SYSTEMS

Typically, an embedded system is referred to as a special-purpose computer system to perform a specific application or a set of activities. Nowadays, embedded systems are broadly used in various target systems from household appliances or consumer equipment to more complicated devices in the field of network or flight control. In this chapter, an overview of embedded systems is given. Then, its hardware and software elements will be described in more details.

4.1 Basic Concepts

Although recent advances in technology have made the definition of embedded systems somehow fluid, it follows some fundamental features that distinguish an embedded system from desktop computers. Embedded systems, for example, are often restricted in resources such as RAM, ROM, or other I/O devices compared to the typical desktop PCs. Besides, they might require power resource such as batteries. Most of them are provided with a simple user interface to interact with the user of the system, if needed. From the software perspective, they have built-in application software that users are prohibited to manipulate. Also, debugging is performed by built-in circuitry [18, 29].

In the early days of emerging embedded systems, they were produced with no operating system. Indeed, a piece of the assembly language program which is stored in memory could accomplish all of the operations of an embedded system. In recent years, customers need embedded systems with higher quality and reliability requirements. Therefore, an embedded operating system has been integrated in embedded system design to provide more complicated services such as multitasking, process and memory management, inter process communication, timers and so on.

Among various kinds of embedded operating systems in the market, Linux distributions have gained popularity as the primary embedded operating system [10].

Embedded Linux is defined as the usage of the Linux kernel and various open-source components in embedded systems. Beside low-cost and availability of Linux source code, there would be some advantages of using Linux as an embedded operating system that convince developers to apply it in their projects. One of the major benefits of Linux that makes it flexible is its modularity. This feature allows developers to eliminate utility programs, tools or other services that are not usable in their embedded system environment. Some other advantages of Linux embedded operating systems are as below:

- **A vast array of development options:** Linux kernel supports every development tools such as C,C++, Java, Fortran and etc.
- **A vast array of free applications:** Hopefully many of free-software applications such as web servers, FTP, Telnet, NTP, SSL, SQL and email exist which are compatible with Linux distributions.
- **Hardware support:** Linux supports many kinds of different hardware platforms and devices. On the other hand, Linux runs on almost all general-purpose 32 or 64 bit architectures like Intel X86, MIPS, PowerPC, IBM and so on [46].
- **Better support for networking:** Linux has integrated the latest network technologies into its kernel so that could support IPv6, IP masquerading, Network Address Translation and so on.
- **The Source code is freely available:** Excluding the payment of license fees, it could reduce the total cost for embedded system and the final product. In addition, it provides the ability to modify, debugging or optimization of the source code for an unlimited period according to system's requirements.
- **Community support:** There are numerous communities and mail lists around the internet that provide the possibility for Linux developers to communicate with each other directly. In fact, they obey the Linux philosophy to share their knowledge to improve the Linux and its features.

4.2 Generic Architecture of an Embedded Linux System

By definition, all embedded systems, as shown in figure 4.1, have a common system model, that is, they have three layers consist of hardware, system software, and

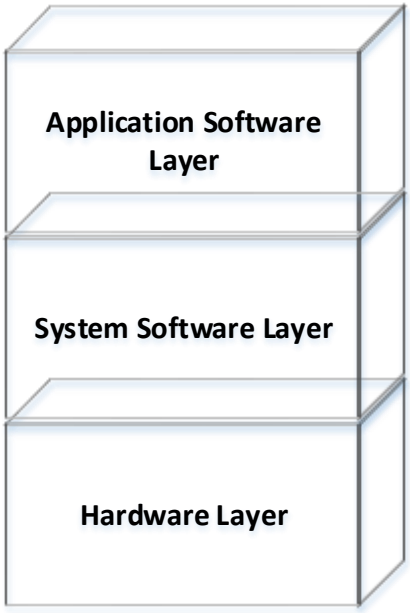

Figure 4.1 *Embedded Systems Model*

application layer. The hardware layer that is an obligation layer in this model contains all the physical devices of an embedded system. In contrast, the two other layers, software and application, are optional. They consist of all of the system software located on the embedded system.

Coming to the hardware details of the embedded system, it is including the following components:

- Central Processing Unit(CPU)
- RAM and ROM
- I/O devices like sensors, keypad, switches,...
- Communication Interfaces such as USB, Serial or parallel port, Ethernet,...
- Power supply like batteries

The software component of an embedded system is concentrated on accessing the hardware resources suitably. This layer consists of both operating system and application software. It is notable that the operating system is not needed in some

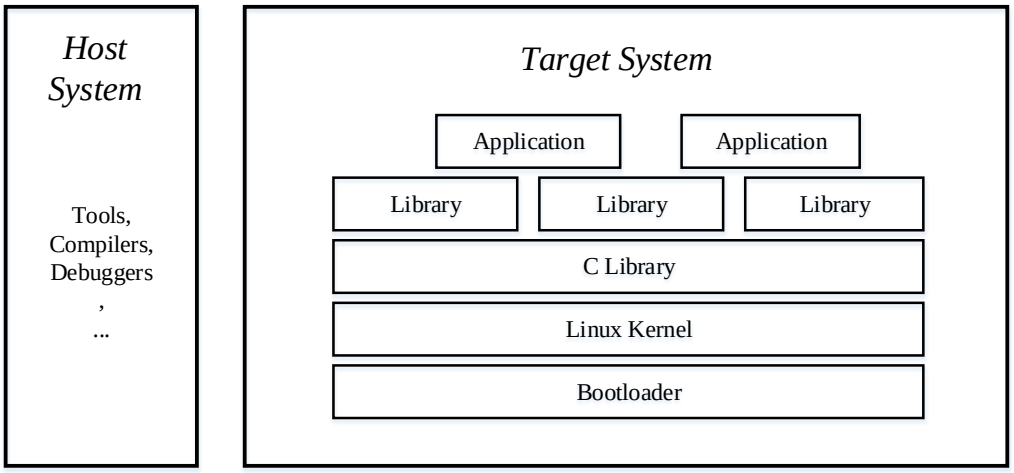

Figure 4.2 *Software Components of a Linux Embedded System*

embedded systems that have restrictions in their design. Indeed, a software routine is written to access system hardware instead of a real OS. On the other hand, a collection of distinct software tools which are settled in a separate computer system is needed to develop the embedded system. These systems are known as host systems which holds all of the running development tools such as editors, compilers, assemblers, and debuggers. In this context, the embedded system is called the target system. The software components of the host and target systems can be seen in figure 4.2.

4.3 Software Elements of Embedded Systems

As mentioned earlier, software elements required for an embedded system are divided into two sections. First, a collection of development tools in the host system which is called a cross-compilation toolchain. The second collection refers to software elements in the target or embedded system [33]. The flow chart for the cross-compilation is shown in figure 4.3. In this model, the source code is developed on the host machine and this code is compiled and linked using the host tools. Then, the code is ported to target system and debugged. If it works properly, it will burn on a flash and will run on the target system, otherwise, it returns to development stage to be corrected.

In the following sections, those software components in host and target systems will be reviewed.

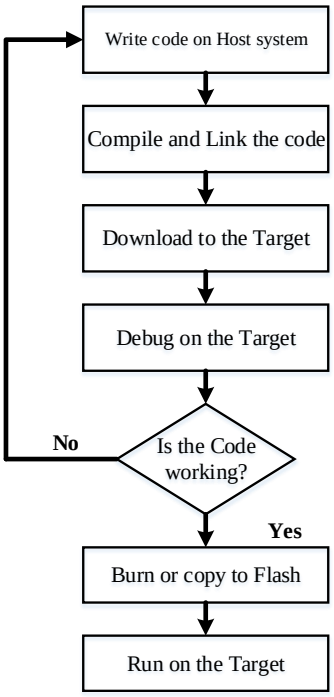

Figure 4.3 *Flow chart of cross-platform development*

4.3.1 Cross-development Toolchain

Toolchain definition: Basically, a toolchain is a set of various software development tools that are linked together with a bunch of libraries that produces the final executable application and provides additional support to build computer software [47]. This process involves cross-compilation, assembly and linking of the generated code.

Why we need Toolchain in Linux: By considering the fact that Linux is an open source operating system, and everybody can access or manipulate it, various teams or projects apply their own requirements and needs to the Linux kernel in order to create a specific application or even a new Linux-based operating system. Therefore, there is no compatibility between variant Linux distributions and development tools in the field of Linux. By contrast, Microsoft designs and develops its own tools with high compatibility with its operating system features and updates them whenever the infrastructure of the Windows operating system changes. Microsoft Visual Studio or .NET platform, for example, are the official software development environment for Windows. But there are no such proper tools for Linux users. Consequently, Linux developers will need to evaluate carefully their required

components and integrate them together in order to have a functional set of tools like the compiler, assembler, linker and so on [50].

Toolchain types: To clarifying various kinds of toolchains, three different machines should be distinguished:

- The build platform: where the toolchain is built
- The host platform: where the toolchain will be executed
- The target platform: where the binary created by the toolchain is executed

For the above three kinds of machines, we can mention four different types of toolchain building processes: [40]

- **Native toolchain**
  - All three machines- build, host, and target- are the same.
  - This toolchain runs on a workstation and generates code for this workstation too.
  - It is available on a GNU/Linux workstation
- **Cross-compilation toolchain**
  - The build and host machine are the same, but the target is different.
  - It is used to build a toolchain that runs on a workstation, but generates code for the target.
  - It is largely used for embedded systems.
- **Cross native-toolchain**
  - The host and target machine are the same, but the build machine is different.
  - It is used to build a toolchain that runs on the target and generate code for the target as well.
- **Canadian build Toolchain**
  - All the three machines are different from each other.

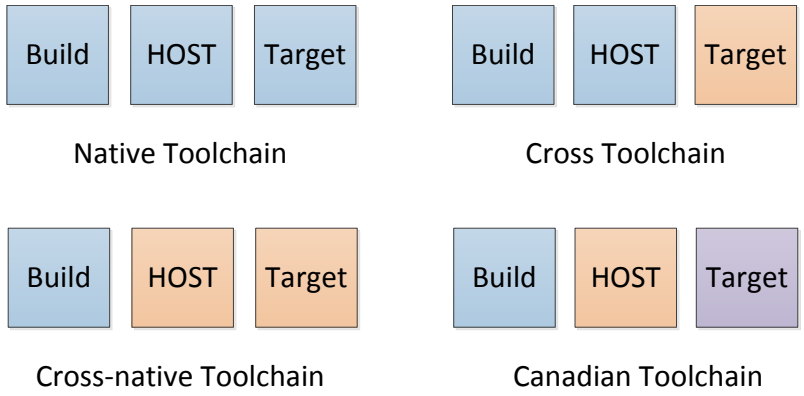

Figure 4.4 *Types of toolchains*

– It is used to build on workstation 1, run on workstation 2 and generate code for workstation 3.

Figure 4.4 distinguishes those Toolchains by colorizing boxes that represent the build, host or target machines.

Cross-compilation toolchain for embedded systems: Since embedded systems are getting gradually smaller, they have restrictions in terms of storage and/or memory that influence their speed and performance. Therefore, it is not a good idea to use a native toolchain for these embedded systems. Instead, developers tend to create cross compilers for their projects so that the limited storage in target machines remains for generated codes and binaries. Figure 4.5 compares the operation of native and cross compiler together in compilation machine and execution one. However, our target machine does not have a primary set of compilation tools; cross compilation uses development tools of the host machine and generates binary for the target architecture.

In the next chapter, the principal components of the cross compilation toolchain will be described in details.

Components of Cross-compilation toolchain

The components of a Linux-based toolchain are GNU Binary Utilities or Binutils, the GNU Compiler Collection(GCC), a C library, and a set of Linux kernel headers for userspace development.

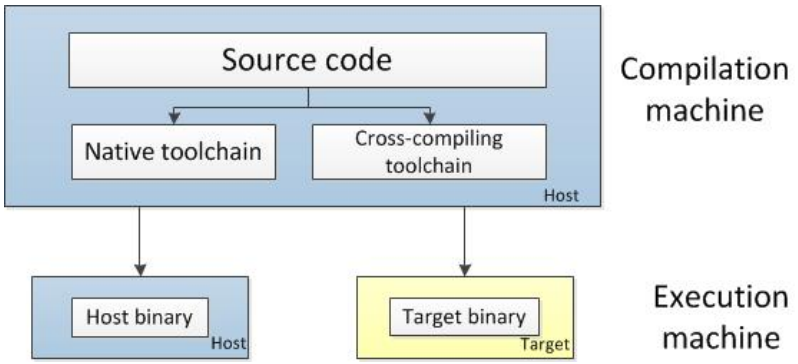

Figure 4.5 *Cross compilation vs native toolchain*

Binutils: The most important component of a toolchain is Binutils which is binary utilities for short. It includes a collection of binary tools used to generate and manipulate binary object files for a given architecture. Although there are lots of useful and practical utilities in this component, the main ones are:[47]

- As: the assembler, that generates binary code from assembler source code
- Ld: the linker
- ar: to create .a archives, used for libraries
- readelf, nm, size, string, objdump, to inspect binaries
- strip, to remove useless parts of binaries in order to reduce their sizes

Compiler: The compiler is the second major component of a toolchain. The GNU Compiler Collection (GCC) is a modern compiler set used by most Linux systems that let you build all kinds of compilers such as multithreading, multilib, shared libraries and so on. GCC comprises a sequence of compilers that enable it to compile different programming languages such as C, C++, Java, FORTRAN, and Ada. It all depends on how you configure the compiler before building it [28, 25].

C library: Is another essential component of a toolchain. It acts as an interface between the applications and the kernel. There are several C libraries like uClibc, gLibc, and eglibC that have their own specifications. Whereas the GCC compiler is compiled against a specific C library, that should be chosen by developers at the time of cross compilation environment. It is not possible to change the C Library component after building the toolchain [32].

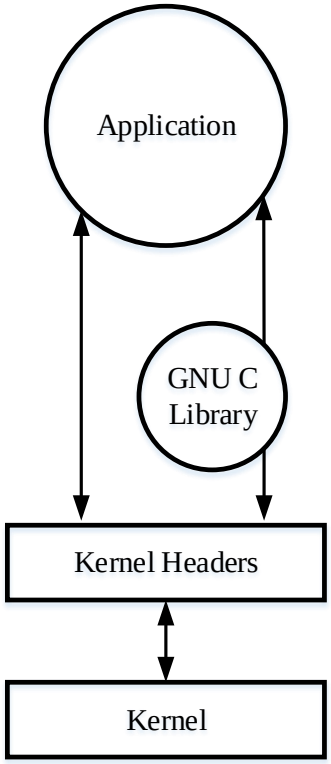

Figure 4.6 *Relation between kernel headers, C library, application and kernel*

Kernel Headers: Kernel headers in the Linux kernel are used for two purposes: [47]

- As an interface between the C library and other applications in user space so that they could interact with the kernel
- As an interface between the components of the kernel

Also, we need kernel headers to compile C library. System calls and their numbers, constant definition, and data structure are examples of kernel headers. Figure 4.6 shows the relation between kernel, Kernel headers, C library, and application.

4.3.2 Bootloader:

Forasmuch as the environmental setting and the underlying hardware are different in desktop computers and embedded systems, their boot process is accordingly different. A desktop computer, for instance, has a hard disk and BIOS. In contrast, an embedded system has flash memory and startup system.

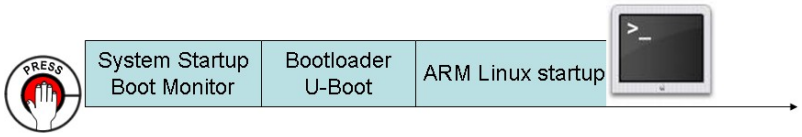

Figure 4.7 *Linux Boot Process*

Generally speaking, the Linux boot process compromised from 3 stages. Figure 4.7 shows high-level view of Linux boot. The first event after powering on the system is executing the BootMonitor system, then the bootloader continues the system boot procedure. Finally, the Linux startup system completes it and the user space applications are ready to be executed.

System Startup:

An embedded system startup is similar to BIOS in a desktop computer that is stored in a fixed location in flash memory. Since it has limited functionalities, it could not load a kernel image. Therefore, a bootloader is needed to complete the process of booting. Although system startup initializes the memory controller and configure hardware peripherals, whose main responsibility is searching, loading, and executing the bootloader program. In the final step, the system startup delivered the control of booting to bootloader while it is loaded into the memory.

Bootloader:

In an embedded system, the very first step to making an operating system running is to boot the system. In a simple word, booting refers to loading the kernel and its infrastructure into memory to starting its execution. This process is performed by the system bootloader. The bootloader is a piece of software code that will be loaded into memory by the system startup. The bootloader usually resides in flash memory and has two main roles: hardware initialization and loading kernel image into memory space. The initialization process differs for each kind of processor, architecture and hardware manufacture. So, it is required to develop a bootloader suited for every platform.

In recent embedded systems, the responsibilities of embedded system bootloaders have divided into two parts as primary and the secondary bootloaders. This possibility makes the bootloader more reliable and provides portability as well. This kind of bootloaders is called multi-stage in contrast with single stage ones [17, 6]. In the following, the roles of these stages is described:

1. **The primary bootloader:**

   It initializes hardware components and allocates memory for the second stage bootloader. Then, it copies the second bootloader into memory space and jumps to the entry point of it. Now the boot process continues in the secondary bootloader.

2. **The secondary bootloader:**

   It initializes other hardware components that are involved in this stage. For instance, it sets CPU speed and clock rate, initializes RAM, disables CPU internal instructions or data cache, configures page sizes and memory management registers and so on. Then the bootloader loads the compressed Linux kernel image into RAM with its proper arguments. The bootloader passes the arguments in the form of tags to the kernel. These tags provide data for the kernel about the situation of the system, for example, size and shape of the memory, also some other architecture-specific structures. Then, bootloader places the root file system image into memory which is used by the kernel in order to load necessary drivers to boot the system. Now the bootloader delivers the control of the boot process to the kernel.

Nowadays, there are many open source or commercial bootloaders like (Das) U-Boot, RedBoot, Lilo and GRUB that are frequently used in embedded systems to facilitate the procedure of boot-up and porting the bootloader to target platforms [17].

4.4 Kernel:

The kernel is responsible for initializing the environment for running C code. Therefore, it executes architecture independent `start_kernel` function, initializes kernel subsystems and loads all user applications [39].

5. PORTING THE LINUX KERNEL TO COFFEE RISC CORE

This section emphasizes on instruction to port the Linux kernel to COFFEE RISC Core that has been introduced in earlier chapters. Basically, the word "porting" could be defined for three different situations: [35]

- Porting to a new board with an already supported processor on it.
- Porting to a new processor from an existing, supported processor family.
- The last one is to port to a completely new architecture.

Since COFFEE RISC Core is designed as a new architecture, the latest case which is porting to an entirely new architecture has been considered. In the following, building the Linux Kernel, toolchain and bootloader is described in more details.

5.1 Toolchain:

As it has been illustrated, the toolchain is the basic component in embedded system development to cross compile the kernel towards the target platform. Therefore, the first step of porting process is creating the suitable toolchain that meets all requirements.

Obtaining a toolchain: There are several ways to get a toolchain: [47, 15]

- Pre-built toolchains:
  - In some cases, there are pre-built ones in the Board Support Package (BSP), which is prepared with the hardware platform by the vendor. As the toolchain is already built and tested by the supplier, it is the simplest and most convenient solution for the developer.

  - The only downside is that you are not able to modify toolchain features such as the type of the C library or ABI.
  - Sourcery CodeBench and Linaro are examples of pre-built toolchains.

- Build a toolchain on your own:
  - Configuring and building an appropriate GNU toolchain by oneself is a complicated process that might take up to several weeks. A person who wants to build a customized toolchain should have a deep understanding of different software packages, their dependencies and the status of their version and needs to investigate errors arising from version dependencies. Therefore, it is a tedious and time-consuming work.
  - Figure 5.1 represents some ideas from the cross tool matrix provided by Kegel website[1].It shows whether the given combination of GCC, gLibc, Binutils and Linux kernel headers can be used to build a cross-compilation toolchain and to compile a kernel for the given CPUs or not.

- Build a toolchain using an automated tool:
  - Due to the discussed shortcomings of the first two methods, the most common solution for building a toolchain is to use utilities that ease the process of building a toolchain with the help of ready made scripts or elaborate systems.
  - These utilities provide shared recipes and patches needed to build a toolchain of desired versions of various software components.
  - They have some built-in shell scripts or makefiles to download, extract, configure, compile and install the components from existing repositories automatically.
  - Moreover, they usually contain several patches that fix issues with the different elements on some architectures. Therefore, they could offer more flexibility not only concerning toolchain configuration, but also in the version component selection that help Linux programmers to omit details of the build process.
  - Crosstool-ng, buildroot, PTXdist, OpenEmbedded, Yocto are the most popular automated tools [15].

[1]http://kegel.com/crosstool/crosstool-0.43/buildlogs/

Table 5.1 *Compatibility between different versions of toolchain components*

	gcc 2.95.3 cgcc 2.95.3 glibc 2.1.3 binutils 2.15 linux 2.4.26	**gcc 2.95.3 cgcc 2.95.3 glibc 2.2.2 binutils2.15 linux 2.4.26**	**gcc 2.95.3 cgcc 2.95.3 glibc 2.2.5 binutils2.15 linux 2.4.26**	**gcc 3.2.3 cgcc 3.2.3 glibc 2.2.5 binutils2.15 linux 2.4.26**	**gcc 3.2.3 cgcc 3.2.3 glibc 2.3.2 binutils2.15 linux 2.4.26**	**gcc 3.2.3 cgcc 3.2.3 glibc 2.3.2 binutils2.15 linux 2.6.9**
alpha	FAIL	Fail gdb fail	Ok gdb ok	Ok gdb ok	Ok gdb ok	Fail gdb fail
arm	Kernel fail	Kernel fail gdb ok	Kernel fail gdb ok	Kernel fail gdb ok	Kernel fail gdb ok	Kernel fail gdb ok
arm9tdmi	Fail	Fail gdb fail	Fail gdb fail	Kernel fail gdb ok	Kernel fail gdb ok	Kernel fail gdb ok
arm-iwmmxt	Fail	Fail gdb fail	Fail gdb fail	Fail gdb fail	Fail gdb fail	Fail gdb fail
arm-softfloat	Kernel fail	Kernel fail gdb ok	Kernel fail gdb ok	Kernel fail gdb ok	Kernel fail gdb ok	Kernel fail gdb ok
arm-xscale	Fail	Fail gdb fail	Fail gdb fail	Kernel fail gdb ok	Kernel fail gdb ok	Kernel fail gdb ok
armeb	Fail	Fail gdb fail	Fail gdb fail	Fail gdb fail	Fail gdb fail	Fail gdb fail
armv5b-softfloat	Fail	Fail gdb Fail	Fail gdb Fail	Fail gdb Fail	Fail gdb Fail	Fail gdb Fail
i686	OK	Ok gdb ok	Ok gdb ok	Ok gdb ok	Ok gdb ok	fail gdb fail
ia64	Fail	Fail gdb fail	Fail gdb fail	Fail gdb fail	Kernel fail gdb ok	Ok gdb ok

After building the development cross-compiler, the target name to be used in later steps is in the form of *coffee-linux*. Moreover, the Linux environment variables need to be modified to include an entry for the cross compiler. For this purpose, we need to define a variable that refers to the executable file of cross compiler and exports it as follows :

- `CROSS_COMPILE` = "/CrossCompilerDirectory/bin/coffee-linux"
- export `CROSS_COMPILE`

Now, the Linux kernel and bootloader can be compiled by the generated toolchain.

5.2 Linux Kernel Modification:

The various versions of Linux kernel source tree are freely available on its official website which is *www.kernel.org*. The latest stable version of the kernel at the time

of writing this document is 4.3.3. Also, there are various patches that are used to fix a specific version's bug. However, it is more reliable to get the desired version by using wget command in Linux terminal. The first line downloads the Linux kernel, then *tar* command will extract it.

- *wget http://www.kernel.org/pub/linux/kernel/v4.3/linux-4.3.3.tar.gz*
- *tar zxvf linux-2.4.19.tar.gz*

The Linux kernel source tree supports various architectures like MIPS, ARM, PowerPC, X86 and so on. In fact, there is a separate directory for each of them, including all hardware-dependent codes regarding that specific architecture. Therefore, we need to have another directory for COFFEE Core too. It would be beneficial for developers to take into account the general structure of one of supported architectures and try to map their own platform with the current ones. In the following, this general rule will be described to make it clear.

In this step of porting Linux, a new directory will be created inside /arch at the root of the kernel tree. Therefor, it should be Linux/arch/coffee in our case. Inside this new directory, the layout is standardized and comprises these sub-directories:

- **boot/:** handling boot process
- **include/asm/:** headers dedicated to internal use, like Linux source code
- **include/uapi/asm/:** headers to be exported to user space, like libc
- **Kernel/:** general kernel management
- **lib:** optimized utility routines, like memcpy(), memset()
- **mm:** memory management
- **configs:** default configurations for supported systems
- **Kconfig:** the main configuration file for the new architecture, it contains arch-specific and arch-independent configuration options

5.2.1 The header files:

The architecture-specific and architecture-independent header files required by Linux are located in coffee/include/ directory. In the recent version of Linux kernel, the header files are divided into two sub-directories, *asm/* and *uapi/asm/*. The first one includes header files that will be used by the kernel interface, and the second header files are invoked by the user interface.

Constituting an enormous number of header files, include directory implementation is a tough task in the procedure of porting to a new processor. For this reason, the structure of header files has been changed slightly in the recent years so that the portion of header files that are common between many processor architectures, has moved to the generic layer of Linux header files that are located in *linux/include/asm-generic/* and *linux/include/uapi/asm-generic/*. In this way, the required generic header files could be referred and accessible through some modification in kbuild file in linux/arch/coffee/include/asm/. For example, in the kbuild of coffee directory two header files trap.h and io.h are referred as follow:

- *include include/asm-generic/kbuild.asm*
- *headers-y + trap.h*
- *headers-y += io.h*

In this step, a set of already implemented header files could be used as a template to make a list of required header files. Then, each of them should be examined precisely to make a decision whether they need to be customized or not. If there are some items that are architecture-specific in a header file, it needs to be modified and settled in include directory of the specific architecture. However, some of the main headers that describe a specific architecture are:

- asm/cache.h: defines cash size
- asm/tlbflush.h: translation lookaside buffer(TLB) management,
- asm/elf.h, defines elf format
- asm/irqflags.h, interrupt enabling or disabling

- asm/page.h, asm/pgalloc.h, asm/pgtable.h, related to page table management
- asm/mmucontext.h, asm/ptrace.h, related to context switching

5.2.2 Boot Procedure:

Another part of Linux architecture-specific code that needs to be developed is the functions of the boot process. As explained in the previous chapter, the bootloader responsibility is initializing hardware and loading Linux kernel image in memory. The configuration of the boot process could be manipulated from Makefile located in *linux/arch/coffee/boot/makefile*. In this file, we can define the name and the format of the output built kernel. The vmlinux, for example, is the raw format kernel which is usable for debugging. Moreover, other variables like ZTEXTADDR, ZRELADDR, `PARAMS_PHYS`, `INITRD_PHYS` and ZBSSADDR would be set that determine different address spaces that the kernel needs to know to operate correctly.

The next step after boot configuration is to write a function in assembly code to define the main entry point of the kernel image. This function is called `start_kernel` that is located in *linux/arch/coffee/kernel/head.S*. Indeed, it determines the bootloader where to jump after loading the kernel image in memory.

5.3 Building the kernel Image:

After kernel code modifications, it is time to build the kernel image in order to load it in the target architecture. For this purpose, we need to enable CROSS_COMPILE entry in the Makefile in (TOPDIR)/COFFEE/boot directory of the Linux tree as follows:

- CROSS_COMPILE = /DIR to the cross compiler
- AS = $(CROSS_COMPILE)as
- LD = $(CROSS_COMPILE)ld
- CC = $(CROSS_COMPILE)gcc

In addition, the LOADADDR variable that contains the address in which the kernel image would be loaded, should be specified in the makefile of the boot directory.

Moreover, the makefile of the top directory of COFFEE architecture, arch/COFFEE/makefile, defines the specifications of the platform, the board and its model that needs to be exported to the Linux kernel:

- platform-$(CONFIG_COFFEE) := COFFEE
- board-$(CONFIG_ALTERA_STRATIX) := altera_stratix
- model-$(CONFIG_RAMKERNEL) := ram
- export PLATFORM BOARD MODEL

After doing all modifications, the kernel has to be configured with *make config* command. Then, the following command would be executed to build the Linux kernel image for COFFEE:

```
make ARCH=coffee vmlinux
```

In our case, that target is vmlinux which is the Linux kernel image. It is worth to know that the linker script under arch/coffee/kernel/vmlinux.lds.S defines the layout of the kernel image that could be modified according to the hardware implementation. This script, for example, instructs the linker how to place the various sections of code and data in the final kernel image.

5.4 Starting the kernel:

The boot sequence always initiates with a small function which is written in assembly language. It is considered as the main entry point of the kernel image and is stored in arch/COFFEE/kernel/head.S. This function indicates to the bootloader where to jump after loading the Linux image in memory. It could be written as follows:

_start:

```
ldra    r1,init_thread_union      //set stack at top of the task union
addi    sp,r1,THREAD_SIZE_ASM
ldra    r2,_current_thread        //Remember current thread
st      r1, r2, 0
ldra    r1, machine_early_init    //save args r4-r7 passed from boot-
```

```
loader
        jalr        r1
        nop
        ldra        r1,start_kernel        //call main as a subroutine
        jalr        r1
        nop
```

As can be seen in the above assembly code, the start_kernel() function which is located in linux/init/main.c has been invoked. Basically, start_kernel() is where subsystems like virtual file systems(VFS), cash, security framework to time management, the console layer and many others are initialized. In fact, it is the first architecture-independent C function that Linux provides. start_kernel() never returns to its caller, since it ends by calling the rest_init() function.

5.4.1 The first kernel thread:

As it is described above, the rest_init() is the last function call which is performed by the start_kernel() function. In this state, the memory management subsystem is completely operational. The rest_init() creates init() threads which obtains PID 1 as the very first kernel thread. The init() thread which is located in /sbin directory, locks the kernel, then it calls do_basic_setup() in order to perform device or bus initialization. After completing kernel initialization, free_initmem() will free any memory that was specified as being for initialization processes.

Once the porting procedure is done successfully, the init process is able to run and give access to a shell. But like other software based projects, the port needs to be maintained or improved. For example, adding support for multiprocessor or implementing more device drivers are the ways to enhance and reinforce the Linux-based system.

6. RESULTS AND DISCUSSION

6.1 Issues:

As the Linux kernel is developed by the Linux communities around the world, there is not much documentation available describing the steps of the porting process on the web. Indeed, lack of valuable documents and materials about Linux programming and the tested solutions for raised unknown errors is hurting Linux developers in their projects. On the other hand, a simple process of porting Linux OS, an MMU-less operating system, may count as little as 4000 lines of code. The code is spread out in different modules of the Linux kernel from device drivers to networking sections. Therefore, getting the Linux kernel running on a new processor is a tough and time-consuming process.

6.2 Achievements:

The present thesis was intended to find a possible procedure that could be followed step by step when porting the Linux kernel to COFFEE Core. Spending countless hours investigating the already supported architectures in the Linux tree, it is discovered that there exists a standard skeleton which is shared by the majority of ports. This skeleton could be divided into two parts logically: architecture-specific code that is executed from the moment the kernel takes over the bootloader until init is executed. Then, the second part of the procedure is regularly executed when the kernel is running normally. In this state, new threads are created, and the OS deals with hardware interrupt or software exceptions, serving system calls, sending or receiving data to user applications, and so on.

In this study, the first part of the porting procedure is covered properly, and the tricks of the implementing process have been described. Also, a customized toolchain software which is a fundamental element of the porting procedure has been implemented. As the COFFEE is not supported by the different components of a

toolchain, it was not possible to use pre-built toolchain softwares which are available on the market. Therefore, the process of building the toolchain has been done from scratch. As this toolchain provides cross-build functionality, the kernel image is built for COFFEE Core too.

7. CONCLUSIONS

7.1 Summary:

This research will serve as a base for future studies on the steps that need to be followed sequentially by the Linux developer in order to make necessary modification in the Linux kernel code so that it could run on the specific hardware platform. The thesis, however, explains the necessary background needed for porting Linux to the aforementioned new platform, COFFEE RISC Core. In the early chapters of the thesis, Linux operating system, and its fundamental components have been investigated briefly. Since the Linux kernel is complicated when it comes to details, the author had a high-level view on this OS and its subsystems. In addition, this thesis provides an overview of embedded systems concepts and components which are vital to know before any implementation.

The worthiness of an OS is evaluated based on how well it could support user processes on one hand, and how well it can implement the services it provides on the hardware. Generally speaking, there are concerns in both software and hardware sides of the kernel while porting Linux to a new architecture that should be taken into account. On the hardware side, however, the concerns are related to the architecture of the new CPU subsystems such as memory management, cashes, and the multiprocessing details. Moreover, other peripheral and I/O systems that will connect to the system needs to be implemented separately. On the other hand, the software concerns are with regard to the compatibility of the user and system software interfaces with the Linux kernel that should be solved as well.

Finally, it is also worth noting that the original version of the Linux kernel is including all modules of a real operating system which in most cases some of them is unusable. In fact, a good design of the kernel subsystems can improve the performance of the whole operating system.

7.2 Future work:

As with all research studies, the analysis and methods presented in this thesis can be extended and improved to give Linux developers a comprehensive guideline. The rest of implementation part of the project could be in the bootloader field that needs to be developed precisely. The kernel image is loaded to COFFEE Core if the bootloader works properly. In this case, the user applications could interact with the kernel by calling system calls. On the other hand, external devices, if they exist, could be integrated into the system by adding their implemented modules in the device driver file of the Linux kernel tree. Each of these mentioned enhancements would enrich this research study.

BIBLIOGRAPHY

[1] COFFEE RISC Core. [Online]. Available: http://www.coffee.tut.fi/documents.html

[2] Z. Bin, J. Q. Chao, Y. L. Xiao, and C. X. Guang, "Research and limitation of system call based on linux platform," in *Electric Information and Control Engineering (ICEICE), 2011 International Conference on*, April 2011, pp. 1592–1595.

[3] D. Bovet and M. Cesati, "Basic operating system concepts," in *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005, ch. 1.

[4] ——, "Memory management," in *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005, ch. 8.

[5] ——, "Process communication," in *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005, ch. 19.

[6] ——, "System Startup," in *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005, ch. Appendix A.

[7] ——, "The virtual filesystems," in *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005, ch. 12.

[8] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Memory Mapping and DMA," in *Linux Device Drivers*. O'Reilly Media, 2005, ch. 15.

[9] P. J. Denning, "Virtual memory," *ACM Comput. Surv.*, vol. 2, no. 3, pp. 153–189, Sept. 1970. [Online]. Available: http://doi.acm.org/10.1145/356571.356573

[10] U. T. Electronics, "2013 embedded market study," 2013. [Online]. Available: http://www.iuma.ulpgc.es/~nunez/UBM2013EmbeddedMarketStudyb.pdf

[11] B. Forouzan and F. Mosharraf, *Foundations of Computer Science*, ser. Introduction to CS Series. Cengage Learning, 2008.

[12] R. Fox, "The linux file system." Taylor & Francis, 2014, ch. 10.

[13] ——, "Managing Processes." Taylor & Francis, 2014, ch. 4.

[14] B. Gerofii, "MINIX VFS Design and implementation of the MINIX Virtual File system," Master's thesis, vrije Universiteit amsterdam, 2006.

[15] A. González, "Application development," in *Embedded Linux Projects Using Yocto Project Cookbook*, ser. EBL-Schweitzer. Packt Publishing, 2015, ch. 4.

[16] B. L. U. Group, "Files: A brief introduction," 2006, accessed: 2015-11-29. [Online]. Available: http://www.linfo.org/file.html

[17] C. Hallinan, "Bootloaders," in *Embedded Linux Primer: A Practical Real-World Approach*, 2nd ed., ser. Prentice Hall Open Source Software Development Series. Pearson Education, 2010, ch. 7.

[18] ——, "The first embedded experience," in *Embedded Linux Primer: A Practical Real-World Approach*, 2nd ed., ser. Prentice Hall Open Source Software Development Series. Pearson Education, 2010, ch. 1.

[19] R. KAMAL, *EMBEDDED SYSTEMS 2E.* Tata McGraw-Hill Education.

[20] M. Kerrisk, "filesystems(5) - linux manual page," accessed: 2015-11-15. [Online]. Available: http://man7.org/linux/man-pages/man5/filesystems.5.html

[21] R. Love, "Memory management," in *Linux Kernel Development*, 3rd ed. Pearson Education, 2010, ch. 12.

[22] ——, "System calls," in *Linux Kernel Development*, 3rd ed. Pearson Education, 2010, ch. 5.

[23] ——, "Process Management," in *Linux System Programming*, 2nd ed. O'Reilly Media, 2013, ch. 5.

[24] J. Mamcenko, "Lecture notes on operating systems." Vilnius Gediminas Technical University, 2010, ch. 2.

[25] T. Noergaard, "Know your standards," in *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Elsevier/Newnes, 2005, ch. 2.

[26] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 4, pp. 247–260, Apr. 2008.

[27] J. P.Kylliainen, T. Ahonen, and J. Nurmi, "General-purpose embedded processor cores the coffee risc example," in *Processor design:system-on-chip computing for ASICs and FPGAs*, 2007, ch. 5, pp. 83–100.

[28] J. Preshing. How to Build a GCC Cross-Compiler. Accessed: 19-11-2015. [Online]. Available: http://preshing.com/20141119/how-to-build-a-gcc-cross-compiler/

[29] P. Raghavan, A. Lad, and S. Neelakandan, "Introduction to embedded systems," in *Embedded Linux System Design and Development.* CRC Press, 2006, ch. 1.

[30] ——, "Real-time linux," in *Embedded Linux System Design and Development.* CRC Press, 2006, ch. 7.

[31] B. Randell, "A note on storage fragmentation and program segmentation," *Commun. ACM*, vol. 12, no. 7, pp. 365–ff., July 1969. [Online]. Available: http://doi.acm.org/10.1145/363156.363158

[32] R. Rehman and C. Paul, "Compilers and assemblers," ser. Bruce Perens' open sources series. Prentice Hall PTR, 2003, ch. 3.

[33] ——, "Cross-platform and embedded systems development," ser. Bruce Perens' open sources series. Prentice Hall PTR, 2003, ch. 8.

[34] D. A. Rusling, "Networks," in *The Linux Kernel*, 1998, ch. 10. [Online]. Available: http://www.tldp.org/LDP/tlk

[35] M. Rybczynska, "Porting linux to a new architecture." Presented at Embedded Linux Conference (ELC), San Jose, CA, 2014.

[36] A. Silberschatz, P. Galvin, and G. Gagne, "The linux system," in *Operating System Concepts*, 9th ed. Wiley Global Education, 2012, ch. 18, pp. 781–826.

[37] A. S. Tanenbaum and H. Bos, "The operating system zoo," in *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014, ch. 1.

[38] L. Torvalds, "Linux: a Portable Operating System," Master's thesis, UNIVERSITY OF HELSINKI, 1997.

[39] A. Vaduva, "Bootloaders," in *Learning Embedded Linux Using the Yocto Project.* Packt Publishing, 2015, ch. 3.

[40] ——, "Cross compiling," in *Learning Embedded Linux Using the Yocto Project*. Packt Publishing, 2015, ch. 2.

[41] F. Vahid, "Embedded systems overview," in *EMBEDDED SYSTEM DESIGN: A UNIFIED HARDWARE/SOFTWARE INTRODUCTION*. Wiley India Pvt. Limited, 2006, ch. 1.

[42] S. Venkateswaran, "Getting started with device drivers," in *Essential Linux Device Drivers*. Pearson Education, 2008, ch. 3.

[43] S. Vermeulen, *Linux Sea*, 1st ed., Sep 2015. [Online]. Available: [http://swift.siphos.be/linux_sea/linux_sea.pdf]

[44] Y.-C. Wang and K.-J. Lin, "Implementing a general real-time scheduling framework in the red-linux real-time kernel," in *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, 1999, pp. 246–255.

[45] K. Wehrle, *The Linux Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel*, ser. Alan R. Apt book. Pearson Prentice Hall, 2004.

[46] K. Yaghmour, J. Masters, G. Ben-Yossef, and P. Gerum, "Basic concepts," in *Building Embedded Linux Systems*. O'Reilly Media, 2008, ch. 2.

[47] ——, "Development tools," in *Building Embedded Linux Systems*. O'Reilly Media, 2008, ch. 4.

[48] Y. Yang, M. Ma, and B. Liu, *Information Computing and Applications: 4th International Conference, ICICA 2013, Singapore, August 16-18, 2013. Revised Selected Papers*, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2013, no. pt. 2.

[49] Z. Yi and J. Peter P. Waskiewicz, "Enabling linux* network support of hardware multiqueue devices," in *Proceedings of the Linux Symposium*, vol. 2, 2007, pp. 305–310. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.798&rep=rep1&type=pdf#page=305

[50] A. Zeichick, "getting started with a linux software development toolchain," May 2012, accessed: 2015-12-24. [Online]. Available: https://software.intel.com/en-us/articles/getting-started-with-a-linux-software-development-toolchain